Tim Inzitari

This project is about the comparison of several different sorting algorithms based on different data sets, and number of data sets. The four data sets are a predefined file input, a randomly arranged vector of integers with N elements, a vector of N elements containing ascending values, and a vector of N Elements with descending values. The Values of N tested are 10,000, 100,000, 1,000,000 elements in each vector.

**Files**

Heapsort.h - Implementation and code for the heap sort function, copied from book as instructed.

Mergesort.h - Implentation and code for the merge sort function, copied from book as instructed.

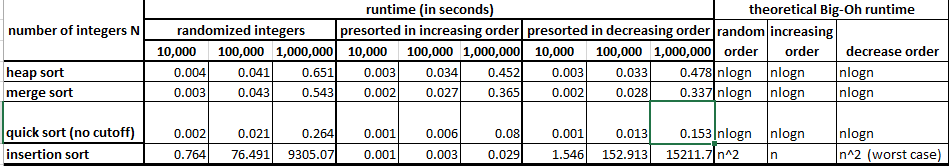
Quicksort.h - Implementation and code for quicksort, modified code inspired by book, to do quicksort throughout the algorithm without degrading into another algorithm. Added to the code was base cases for only having 3, 2 and a singular elements in the arrays.

Insertion.h - implementation and code for the insertion sort function, copied from the book as instructed

Sorting.cpp - Driver file for testing algorithms and analyizing run times. Contains Main Function and calls all the functions.

Project3.pdf - This file containing my report.

**Runtime Analysis and Comparisons**



**Runtime Analysis of Each Function**

*Heap Sort*:

Heap sort is sorting based on the binary Maximum heap data structure. Which would take O(N) time to build. It then sorts the array by doing a Delete max and returning the “deleted” max item to the “last point of the array” pointer location, swapping with current value. Then decreasing the “last data point of the array” pointer by one memory location of the array. It then heapifies the newly sized array. This time takes O(logN) time and it does it N times for each of the N elements… This makes the Total runtime to be O(N+N\*logN) which simplifies to **O(NlogN) in all scenarios**, whether or not the array is presorted or not. Which explains why all the run times for Random, presorted ascending and presorted descending is all around the same value, for their respective element size. It also explains why when the size of the data set is increased the time is not heavily punished do to the growth rate only growing at a NlogN rate.

*Merge Sort:*

Merge sort is a divide in conquer algorithm that is at **O(NlogN)** time, the input it repeatedly halved, which takes O(logN) time and then this happens N times so it is O(nlogn) time, this explains why the times are the so similar same no matter the data set (random takes a little more time as more diverse merges take place that access both arrays), whether random, presorted ascending or descending the time is the same for any N amount of elements. Which explains why in my test times are near the same for all data sets on the merge sort.

*Quick Sort:*

Quicksort is a recursive algorithm that runs average **O(NlogN)**  time. Its worst case is O(N2) and typically only occurs when a poor pivot is chosen, due to the median of three pivot selection we use in the algorithm this very rarely occurs. Some scenarios for it to happen will be if the three higher/lowest items in the set are chosen as the 3 for every occasion, which is highly improbable. This explains why all run times are so fast and appear to be close to O(NlogN) time of the heap and merge sort, (Quicksort is actually faster due to the nature of the algorithm.) All the run times are near the same, with randomized being a tad slower than presorted arrays, which can make sense, however the difference is negligible compared to the other types of sorts.

You can see the speed of the quicksort algorithm in how the time change is so mild as the size of the set increases, with none of the sets taking more than a quarter of a second at most. This is explained because the quicksort is the fastest comparison algorithm, the inner loop of the algorithm consists of an increment/decrement (by 1) a test and a jump. There is no extra juggling as there is in merge sort which contributes heavily to the speed.

*Insertion Sort:*

Insertion sort is a simple sorting algorithm that consists of N-1 elements, for pass p =1 through N-1 it ensures that elements in positions 0 to p are sorted. This algorithm runs in Average case **O(N2)** time. The best case runs in O(N) time and occurs when the array is already in sorted ascending order. The worst case is when the array is sorted in descended order and every pass consists of an addition to the beginning resulting in maximum number of array movements, this causes the run time to be a O(N2) that takes the full N2 time. This can cause for some massive run times in the set. These run times explain why presorted ascending order takes incredibly fast run times, but random and descended orders take a ridiculous amount of time. This is due to the fact that the N2 growth rate causes each factor of 10 to increase the run time 100-fold. Which grows at an incredible rate, (1.5 seconds in 10k for descending order, to 152s in 100k, to 15211s in 1million.)

**Comparisons Between Algorithms**

*Quick Sort and Insertion Sort*

The most drastic difference in runtimes appear with quick sort and insertion sort times. While insertion sort is faster in their best case of presorted ascending data, due to the run time being O(N) compared to quicksort’s O(NlogN). Also in small size arrays (N <= 20) Insertion sort performs better than quicksort, and due to quicksort’s recursive nature this will occur fairly often. The good news however stops there for insertion sort.

For random and descending ordered sets, to call the quick sort faster than insertion sort would be an understatement. When compared the insertion sort is as slow as tectonic plates. For randomized integers the quick sort took 0.002s for N=10,000 while Insertion sort took 0.764 seconds and it only gets worse from there, as Log10(N) increases by 1 the insertion sort takes 100x longer due to the N2 nature, while Quick sort grows much slower at a NlogN pace. For 100 thousand elements, Quicksort took 0.021s while insertion sort took a minute and 15 seconds. For 1 million elements, Quicksort took 0.264 seconds while insertion sort took over two and a half hours, ( I simply ran this one while I slept). And it is only worse for when insertion sort is at the worst possible case of reverse sorted datasets. With for 1 million elements Quicksort taking 0.153 seconds, Insertion Sort took over four hours.

While for smaller data sets (or sets that are already sorted) insertion sort can keep up with or be faster than quicksort, as the size of the set grows, the runtime of Insertion sort grows so incredibly faster than quicksort, that one could begin to question why you would ever use insertion sort.

*Heap, Merge and Quick Sort*:

Heap, Merge and QuickSort all run on simplified O(NlogN) time. However if you take a look at the books analysis’s of each algorithms runtimes (Heap: pgs 301-304, Merge: pgs 306-309, Quick: pgs 318-321) you can see that the expanded runtime formulas are slightly different. For example, on page 304 it states that Heapsort will average case a 2NlogN – O(N) comparisons, which while incredibly fast compared to Merge and Quicksorts equation of T(N) = cNlogN + N. Which resolve in faster runtime and explains the slight time difference and slowness of heapsort compared to the others.

It doesn’t however explain the difference in times between Merge and quicksort that result in quicksort being much faster than merge. This time difference is explained by the fact that the c in the T(n) = cNlogN + N equation means different things in each algorithm. In merge sort, c refers to the time required to merge two arrays. While the c in quick sort refers to the partition time of two arrays, since Partitioning takes significantly less time than merging, the quicksort ends up having a faster algorithm.